Getting started with (modern) Python packaging

Filipe Laíns (@FFY00)
@MissingClara / @MissingClara@gensoukyou.jp.net

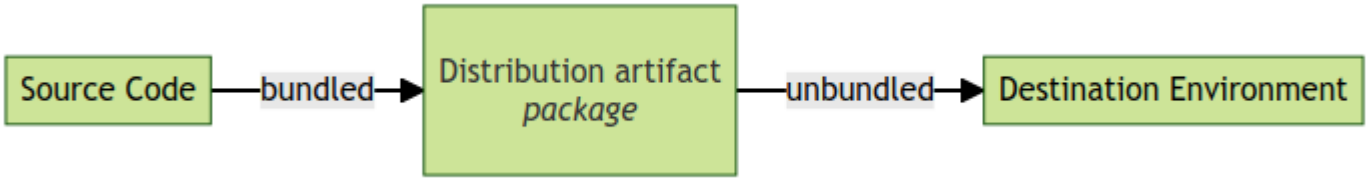
About Me

- Involved in Python packaging for 3 years
- Involved Linux distribution packaging for 5 years

- Upstream maintainer for:
  - pypa/build (as in *pip install build*)
  - Python (sysconfig, import system, etc)
  - Arch Linux (Python, hardware dev tooling, etc)

What is packaging?

- What? A solution for sharing code between people.
- How? By bundling into something you can easily share with others.



What is required?

- A distribution format
  - Usually it consist of a single file (often a ZIP)
  - Contains your code
  - Generally includes some metadata (project name, version, etc.)
- Extras
  - A package repository (peferably standardized)
  - Etc.

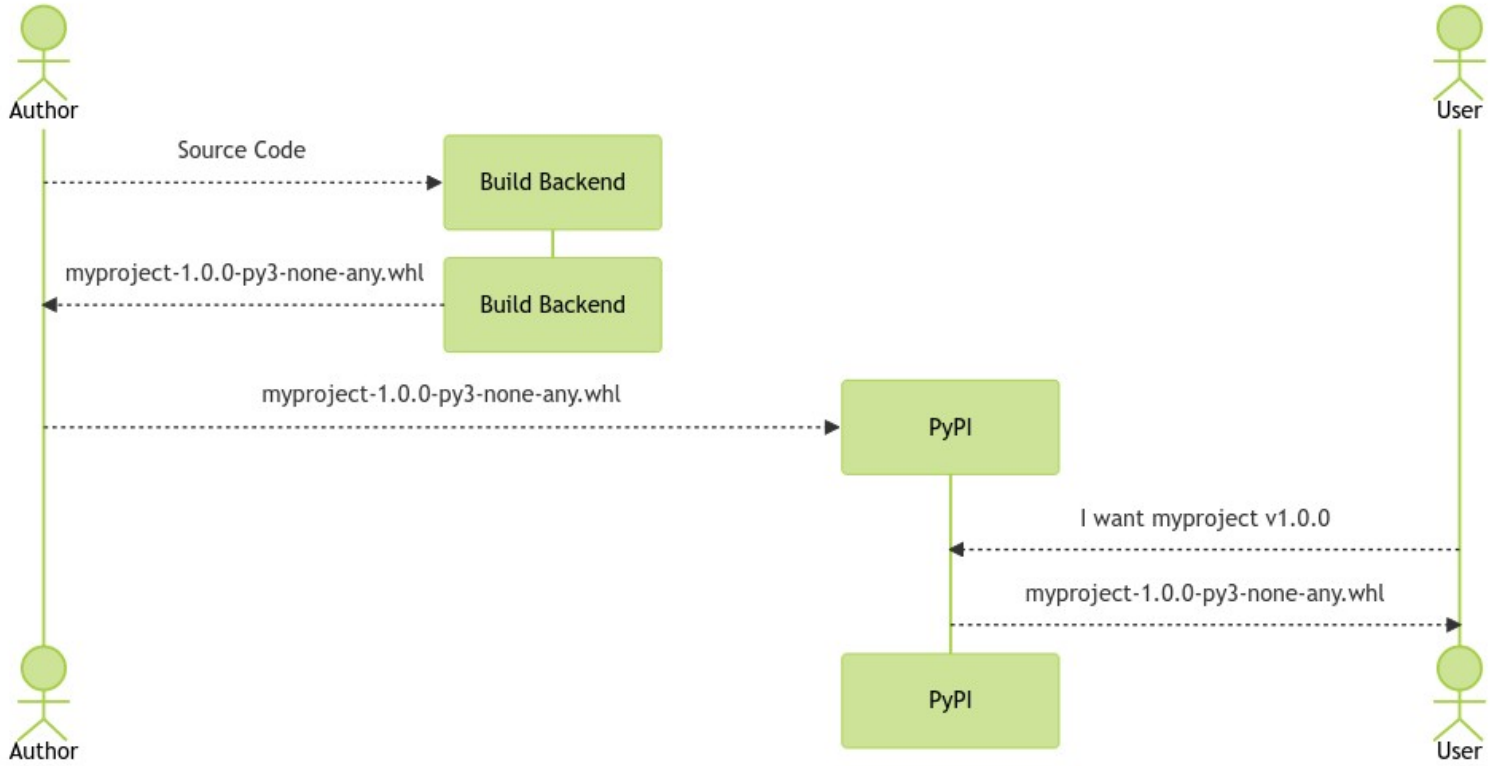
What does it look like in Python?

- Distribution format
  - Source: sdist (.tar.gz – uh... usually)
  - Binary: Wheel (.whl)
- Package repository: PyPI (Python Packaging Index)

How, but I don’t know how these things work...

- You don’t need to!
- We have something called *build backend*
- Build backends implement this for you

Releasing a package



The old way

- You have probably seen a setup.py file
  - This used to be how you packaged Python projects
- Some (very simplified) history
  - *distutils* was added to the Python standard library
    - Slow to update because it's tied to the Python installation
    - Could not support all possible use-cases
  - People started write alternatives outside the stdlib
  - After a while *setuptools* emerged as the de-facto choice
  - *pip* was created and effectively solidified *setuptools* as the only choice

Modern Python Packaging

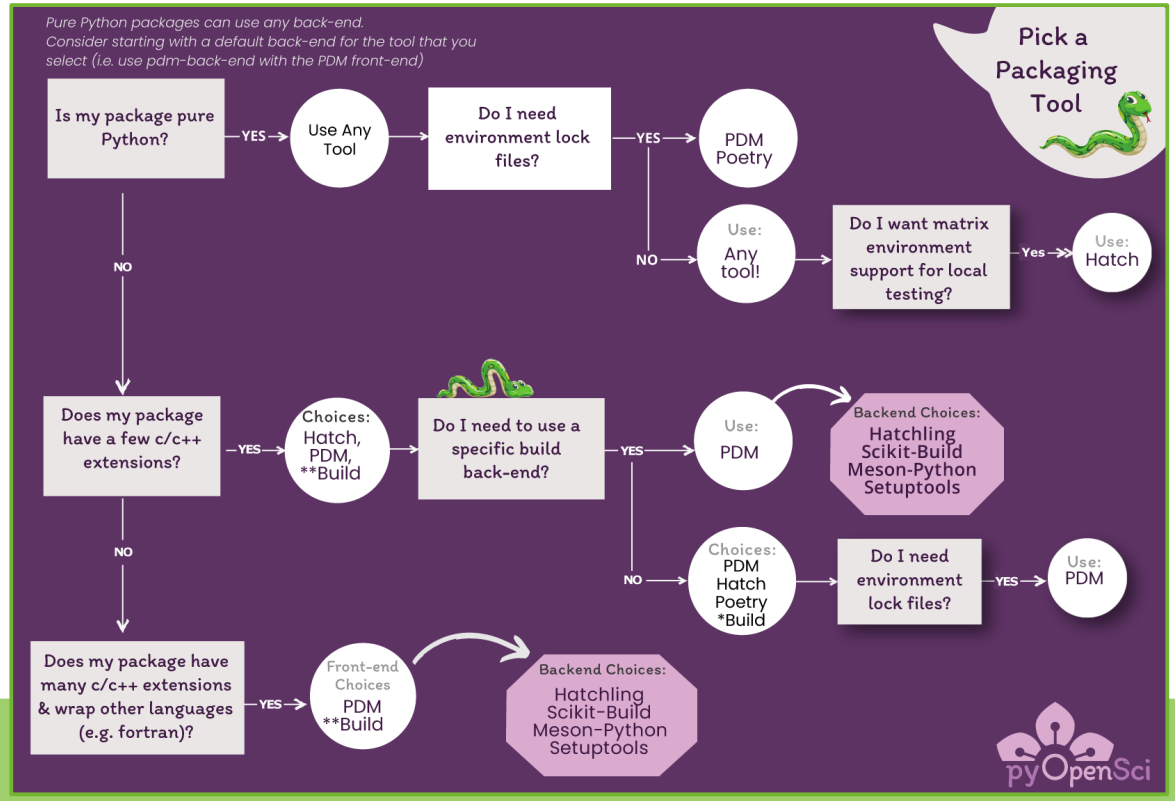
- Standardized an interface for build backends
  - Now anyone can write their build backend
  - We are no longer limited to *setuptools*
  - It made it easier to write new package managers and other kinds of packaging tooling

Getting started

- Choose a build backend
  - setuptools (good for general purpose)
  - flit (pure-Python only)
  - hatch (more integrated dev experience)
  - poetry (more integrated dev experience)
  - pdm (more integrated dev experience)
  - maturin (for Rust)
  - meson-python (for using Meson as the build system)
  - Scikit-build (for using Cmake as the build system)
  - Etc.

Guidance (for choosing a build backend, etc.)

- I recommend looking at the pyOpenSci packaging guide



Creating a project

- Look at the documentation for the build backend you are using
- Create a *pyproject.toml* file (at the root of your project)
- Specify the build backend and build dependencies

```
[build-system]
requires = ['hatchling']
build-backend = 'hatchling.build'
```

Creating a project (cont.)

- Add the project metadata
  - There’s a standardized way to specify project metadata in pyproject.toml
    - See *Declaring project metadata (PyPA specifications)*
    - Not all build systems support it!

Creating a project (cont.)

```
...

[project]
name = 'myproject'
version = '1.0.0'
description = 'Our example Python project'
requires-python = '>=3.8'
license = {file = 'LICENSE'}
authors = [
  {name = 'Filipe Lains', email = 'lains@riseup.net'},
]
```

Sharing the project

- Building
  - (install pypa/build) *pip install build*
  - *python -m build*
- Uploading it to PyPI
  - (install twine) *pip install twine*
  - *twine upload dist/**
    - To upload to TestPyPI you can pass the *-r testpypi* option

Note: Some build backends (eg. *flit, hatch, poetry*) have their own CLI, which may support these fatures (and even more).

Thank you

- Questions?
- Slides available at https://github.com/FFY00/talks
- Feel free to find me afterwards if you want to talk about packaging 😅